UniVista
Graphical Interfaces for Scientific Modeling Codes

Edward Peterlin, '99

Advisors:

JP Singh
CS Department

Daren Stotler
PPPL

Russell Hulse
PPPL

This paper was written in accordance with University regulations.

Edward H. Peterlin
May 3rd, 1999

UniVista

Prior Research

UniVista arose as a continuation of a prior UniVista project at the Princeton Plasma Physics Laboratories (PPPL) in 1996. This project aimed to create a generic user interface tool for scientific modeling codes to help reduce the learning curve associated with them. This project was begun in Java 1.0 and was hampered by the immaturity of that platform. The person in charge of developing this early UniVista left the PPPL, however, and the project was effectively shelved. Its objectives were not forgotten and were incorporated into this project.

Other research has also inspired UniVista. In 1995, the Amber project of the PPPL aimed to create a universal dataflow description of modeling codes. Most of the time in this project was spent trying to architect the interfaces to be universal, not in actually implementing them. Through this project it was shown that the basic problem in creating a universal tool is data management. This recognition led to the flexible design of UniVista as well as motivating the usage of a database to store information about modeling codes.

The PEST (Princeton Equilibrium and STabilitiy code) GUI developed by Manickam at the PPPL in 1998-1999 illustrates that the approach of developing a forms-based GUI for modeling codes is attractive. The PEST GUI, shown in Figure 6, has been in use for a while and seems to provide a comfortable environment, even for experimentalists. The PEST GUI is limited, however. It is implemented in IDL, requiring third-party labs to purchase the IDL environment before being able to use it. In addition, it is specific to the PEST code and is not generalizable for use with other codes. Furthermore, even adding new PEST variables requires specific knowledge of IDL design tools. However, the PEST GUI does demonstrate that user-friendly graphical interfaces for scientific modeling codes can be designed.

Why UniVista?

One of the problems with modeling codes is that the traditional namelist or formatted Fortran write style input file is not the most user-friendly way to interact with the program. These text only files contain lists of numbers or names and values, with a potentially free-form structure. There can be hundreds of variables for a given code that can be modified in a single file. The modeling codes also have complicated behaviours and usually lack thorough documentation describing what each variable controls and what its allowable values are. For all except the most experienced users of a particular modeling code, this can present a problem. Beginning users are frustrated at the steep learning curve associated with modeling codes, and experienced users who can't easily contact the code's creators may be frustrated when something doesn't work or isn't explained. As modeling codes begin to be shared amongst laboratories for cooperative research, this problem will be exacerbated.

Scientific users also have specific ways in which they utilize modeling codes. Frequently users won't change every single variable at once, but rather only modify a few specific ones to study a certain effect with the modeler. When performing a particular study, users tend to limit the number of changes they make from run to run. This allows users to do focused research and is a common usage pattern.

Modeling codes also frequently lack good documentation. Currently, the most thorough and reliable documentation for a modeling code is the author of the code. Authors tend to not provide great written documentation, and users rely on personal contact with them to get explanations. As codes get distributed outside labs, authors will be unable to provide extensive support. With the lack of documentation, outside users will become frustrated at trying to learn the complexity of the programs.

The current state of modeling codes is the result-oriented nature of the research and the need for continual, significant changes to the codes. The primary objective for most code authors is not the code itself, but the research publications that will follow from its results. Hence, these codes typically do not have polished interfaces or extensive documentation. The authors need tools which will allow them to quickly design and modify better interfaces for their codes.

UniVista is designed to address these issues. UniVista provides the modeling code author an easy tool with which he can create easy-to-use graphical interfaces, with built-in documentation, for his program. UniVista assists in the traditional process of research by allowing users to create specific studies that focus on the variables they want. Users can easily make new runs that only change a few things from a prior one, and everything is automatically archived. By storing the majority of the data UniVista uses in a multi-user database, interfaces, studies, and results of particular runs of the modeling code can easily be shared by all users. The combination helps to aid new and old users alike and provides a collaborative research environment.

Designing UniVista

What is UniVista Designed to Do?

UniVista is designed to be a graphical user interface for scientific modeling codes, encompassing all the steps of authoring and utilizing an interface. An interface tool for the scientific modeling environment needs to respond to the special requirements of this environment, especially organization of large projects, backwards compatibility with existing modeling codes and methods of use, and integration with research methodologies.

UniVista addresses organization in many different ways. One addresses the hundreds of variables that can be contained by a given modeling code. UniVista gives the designer the ability to organize the variables onto different screens, isolating groups of related variables as well as emphasizing and providing quick access to important ones. Another organizational problem with large modeling codes is being able to keep track of all of the detailed documentation. Notes on how to properly use a code may extend down all the way to the variable level, such as for description of valid values. UniVista provides the author with the ability to create full online documentation for the code, with both an internal help description as well as a URL associated with each item of the interface for more detailed documentation. UniVista also helps keep track of past results. With most modeling codes being namelist based, the user must do all of his archiving manually. Users may not retain complete archives due to the maintenance cost needed to manually maintain the archive. UniVista assists this problem by providing an automatic archiving feature for each run a user performs. The input variables are stored in a database along with any output the user desires. The user can then look up in the database later on to get the results of his past runs.

Backwards compatibility is another issue being addressed by UniVista. Most modeling codes are namelist based, and thus UniVista needs to be able to translate its internal interface into a namelist that can be run through existing modeling codes. UniVista should also have the ability to do minor ksh scripting to allow all of the legacy preprocessors to be invoked automatically. Users who are currently using modeling codes, however, are accustomed to working with namelists. UniVista must support this method of work so that incorporating UniVista into their project requires no modifications to the modeling code. By providing import and export capabilities, UniVista will not only be able to create namelists for the codes but also be able to read in a user's existing namelist and populate the internal interface with the data. The data from these namelists are then imported and archived as any other data in the program are treated. By containing this import functionality, UniVista attempts to co-exist with namelists and not replace them.

The interface of UniVista is also designed to adapt to the ways in which researchers use modeling codes. Frequently when working a researcher will be focusing on one aspect of the modeling code. UniVista's interface creation tools allow designers to create interfaces featuring specific sets of variables while setting defaults for others. Researchers may also encounter results that display a behaviour they wish to explore further. To this end, UniVista allows interfaces to be created from specific runs of the modeling codes, allowing all the variables to be set up as in that run. This allows researchers to then modify a few key variables from there and explore the behaviour more.

These three factors of backwards compatibility, adaptive interfaces, and organization were kept very much in mind when developing the design of the software. By focusing on these three important needs of researchers in the fundamental design, it is hoped that UniVista will prove to be useful as well as be adopted by its intended user base.

Fundamentals

There is a certain fundamental terminology that is used throughout the description of UniVista's design encompassing interface elements, concepts, and user levels.

UniVista is primarily designed to provide a tool for building graphical user interfaces for physical simulation packages. The division between the description of the simulation program, the interface, and the use of the interface to run the program is split into the UniVista concepts of codes, studies, and runs.

A *code* contains all of the information describing a particular version of a simulation program. This includes descriptions of all of the variables (including default values, short documentation, URL for further documentation, consistency checks, format, and preferred editor), the input and output file formats, and commands that need to be run on a remote machine to invoke the program.

A *study* contains a user interface design for working with a particular code. The study consists of a group of *screens*. Each screen is a window that will eventually be presented to the people using the study to input and examine variables. The screens contain variables, graphical and static text elements, and links to other screens. A study can also have overrides for the variable defaults which supercede the code defaults, but only for that study, as well as short documentation.

A *run* is what corresponds to an invocation of the user interface defined by a study to generate files (e.g., namelist text files) and execute the simulation software.

There are also different user levels within UniVista for defining these different types of objects. The user level is set on a database granularity, that is, users can have different capabilities depending on their user profile in each database they use.

The *code author* is the person who has the ability to add a new code to UniVista. This person is adding the ability for UniVista to work with a new simulation package, or a new version of an existing one. This person should be familiar with how the simulation package interacts with its input files and should be able to describe the variables of the simulation program. This user is defining how UniVista interacts with outside programs.

The *study author* has the ability to create and modify studies within UniVista. They are limited to creating interfaces for pre-defined codes. The study author can set new defaults and documentation for variables that are visible within that study only. They can also control the screen appearance and screen hierarchy of the study.

The *run author* is limited to using pre-existing studies to input specific information about the variables. These people can create runs from studies, or they can create runs from using the initial values of variables of a previous run.

A *tree control* is a hierarchically arranged tree of items. Different levels can be opened and closed by clicking on boxes to the right of each level of the tree. This is similar to the tree control found in the Windows Explorer or the twist down lists for folders in the Macintosh Finder. Drag and drop is used extensively both between windows, but also within a tree control in many places. By dragging an item into a new level of the tree, it can be moved in the hierarchy.

Starting Up

The first thing that a user needs to do when starting UniVista is connect to a database that contains the data files for UniVista. This is done by selecting the database server from the list of databases as shown in Figure 1 and clicking “Connect.” A new database server can be added to the list through clicking “add database,” which brings up a window where the user can type in the address of the server. The control presently is a tree since in future versions it is expected for some type of name server to be available to list all of the accessible UniVista database servers within a particular high level domain such as "pppl.gov". In the first version, however, the user will simply be able to create folders to categorize the servers they want to use (no auto-searching).

After a user connects to the database, they will be prompted for their login and password for that database as shown in Figure 2. After the login phase is completed successfully, UniVista will load the user's privileges (user level) for that database server, indicating whether they can create or modify certain screens of the study.

After the user logs in, they will next be presented with the main menu shown in Figure 3. The main menu is simply a group of buttons that allows the user to choose what they want to do first. The user can switch between any of the modes that they have privileges to do at any time from the main menu bar, so the initial choice is non-binding.

The choices are mostly explained in the later sections, with the exception of the program settings. This button is used to allow the user to change preferences of UniVista such as variable types, default servers, automatic connections, and user/password saving.

Variables

A variable in UniVista is associated with one of the parameters of a modeling code. Variables in UniVista have the following fundamental types: 4 byte integer (short), 8 byte integer (long), IEEE floating point (float), IEEE double (double), float real/imaginary complex pair (complex), double real/imaginary complex pair (double complex), logical flag (boolean), and packed array of characters (Hollerith).

Many different qualities are associated with a variable. Some of them that are specified cannot be overridden in studies. These are usually things which the code depends on to get the output namelist correct. These include the variable name, the fundamental variable type, and its output format. The settings in the code are used if the study author or run author does not change them. These qualities include the default value of the variable, short description of the variable, URL pointer to further documentation, JavaBean used to display the variable’s value, and consistency checks. These can be modified to fit the particular way in which a variable is being displayed and changed in a study.

Consistency Checks

Inside UniVista support is provided to allow small pieces of code to be run before creating a namelist for dispatching to the modeling code. These small pieces of code are called consistency checks.

Expressed in a language that allows for simple arithmetic operations, comparison and conditional constructs, and assignment, these pieces of code are intended for two purposes. One is to be a quick check on the value of a variable to ensure that it is sensible for the modeling code or a study. By performing simple checks, UniVista can help alert users unfamiliar to potential side effects of a particular variable inside of a modeling code.

The second use for this code is to be able to create preprocessor-like snippets to alter the values of other variables based upon the result of an expression. This allows on a simple level for switches to choose between various preset parameters. They could also be used to help correct errors as well.

Consistency codes are only run once on a variable in an undefined order. This means that circular assignment constructs such as “if a>3 let b=1” “if b<2 let a=4” will have unpredictable results. It is up to the code and study authors to avoid using circular assignments.

Creating a New Code

When a new code is created, the user must input the variables and other information needed by UniVista to appropriately generate the output files and run the actual program. When a new code is created, the user is presented with a dialog shown in Figure 4. In this dialog, the user can see a list of the variables on the left and the properties of a selected variable on the right. In the buttons underneath the tree list of variables, the user can create a new variable grouping or a new variable. Variable groupings are like folders; they help the user to organize the large number of variables for a particular code. On the right hand side of the screen is a properties area with buttons that allows the user to change the default short description of the variable, the variable name, the URL location of extra documentation, default values of the variable, and any consistency checks that need to be done for it. Underneath the Properties area are two buttons that allow the user to commit any changes they have made to the variable or to revert to the old values.

The variables are listed in a tree on the left that allows the transfer of a variable into a new group by dragging its name into one of the group headings within the tree.

In the menu bar for the code editor will be an Import command to allow the user to import a formatted namelist of values to use. Under the menu bar a Find command will also be located to allow a variable to be found by name or by description contents. These two commands will appear in a consistent place as they are shared by many of the parts of UniVista that use variables.

After the variables have been completely specified, the user can hit the "finish code" button, and UniVista will create all of the appropriate tables in the database. After this creation, UniVista will then generate a default list of screens. Each group will have its own screens with buttons linking to any groups that it contains. On the screen will be a standard layout of the variables that are contained in this group. After these screens have been created and inserted in the database, the study editor component is opened on this default study to allow the user to customize its appearance. The default study is created using the "terse" layout engine.

The Study Editor

The Study Editor is the component of UniVista that is used to create the graphical user interfaces for particular codes. It is highly graphical in its own nature, allowing users the flexibility to create multiple styles of studies.

There will be a design area which reflects exactly what the run user will see when he creates a new run from the study. Inside of this design area the variables and other objects will be present. When one of these objects is selected, information about it can be changed using the Property Editor (see below). The object can also be dragged around inside of the design area to reposition it within the screen. When the object is selected, standard Edit menu commands can also be performed on it to copy, cut, or delete it. The design area will also feature an auto-snap and auto-grid feature. *Auto-snap* means that when a variable's name and input area are dragged close to a boundary of second variable, the variable being moved will align itself to the edge of the second variable at a preset distance away. This allows for easy relative arrangement of variables and objects. *Auto-grid* means that objects can only be placed at the intersection points of an invisible grid of evenly spaced lines. The user can have control over the width of this grid. Having no auto-grid snap is equivalent of allowing objects to be placed at the intersection point of a grid where there is 0 spacing between the grid lines.

In addition to the design area, there will be several palettes that float above it. The *variable list palette* (see Figure 5) shows all of the variables that are defined in the code. The tree contains the variables in the group listing as defined by the code author. This listing is immutable. To find a specific variable in the group tree the user can click on the Find button to search for a variable based upon its name or its appearance. The button below it, "Make Alias", is enabled only for variables of array types. This allows a study editor to create an alias of a particular row, column, or entry of an array. This is placed under a special "Aliases" group at the top level of the tree list. The study author can then use this alias variable to place a UI element on the screen for that particular subset of the array only. When a variable in the list is clicked, its default values and default one-liner can be modified using the Property Editor. To add a variable to a particular screen, the user simply drags the variable name from this list into the design area and the variable will be added. If a list of variables is dragged to the entry area, they will be added using the terse layout engine. In the list of variables, variables that have any properties from the code overridden will appear in red, variables that have been placed on a screen in the study will appear as normal, and variables that have not yet been placed on any screen will appear in yellow.

There is a second palette, the *unplaced variable palette*, is similar to the variables palette (see Figure 5) but without the Make Alias button. It contains the variables of the code (in the group hierarchy as the code author defined) that have not yet been placed on any screen in the study. This list gives the study author quick access to changing the defaults for invisible variables by selecting their name from this list to update their properties in the Property Inspector, as well as quick access to placing them on the screen through drag and drop.

The next type of palette is the *screen hierarchy palette*. This palette, shown in Figure 5, can occur multiple times. At the top of the palette is a popup menu showing which study's screens are being listed in the tree control below it. When it says "current", it shows the list of screens for the study being currently worked on. The popup menu contains a list of all of the other studies for the particular code being used. If one of them is selected, the tree will contain the list of screens for that other previously created study . Beneath the list there is the ability to find a screen based on its title as well as the ability to create a new group. The screens can be structured into groups in the same fashion as the variables when codes are being created. They can also be reorganized between the groups in the same way. By clicking on a screen to select it from the tree, the Property Editor can be used to change the help string and the screen title. This can only be done for screens in the "current" screen list palette for the study being edited. If the user drags a screen from the "current" list into the design area, a screen link button will appear on that screen being edited that links it to the screen that was dropped. If you drop a screen from a list of screens from a different study, that screen will first be copied into the "current" study and then be linked. If you double click a name in the "current" list, the design area will shift to edit that screen. If you double click a name in the list for a different study, you will be prompted to see if you want to copy the screen. Dropping a screen name dragged from a list of a different study onto the list of the "current" study will cause that screen to be copied into the "current" study being edited. Multiple instances of this palette can be created through menu commands.

The next type of palette is the *tools palette*, shown in Figure 5 as well. The tools palette allows the study author to choose which tool they want to use in the design area. The tools provided include the arrow for selecting, moving, and resizing objects, a text tool for typing static text, a button tool for linking to other screens, a rectangle tool for drawing rectangles, and a line tool for drawing straight lines.

The *property editor* is a window that is by default empty. The contents of the property editor change depending on which object is currently selected. If the user last clicked on a variable from one of the variable lists, the default value, consistency checks, and one-liner for that variable can be changed. If the user last clicked on a screen from one of the screen list palettes, the user can then change the screen's title and help string. If the user last clicked on a variable input area inside of the design area, the user can change the default value, consistency checks, and one-liner for the variable along with adjusting its appearance on the screen including properties of the JavaBean.

The study author uses these palettes in conjunction with wizards to author the graphical interfaces of studies. In the menus along with these include an import command for setting default values for variables from a namelist, including any aliased variables. The study author can also choose to use a wizard that takes lists and groups of variables and automatically constructs screens using the layout editors. Studies can also be created by cloning an existing run and using its variable values as the study defaults.

After the study has been finished, any unplaced variables are automatically put onto a set of *study default screens* laid out using the terse layout editor. These screens are read-only for run authors. They simply display the default values of the variables for easy reference by the user. To change one of the default values of these variables, the study author must edit the study.

Standard Variable Appearance

Each instance of a variable on a screen will have a standard interface that is present regardless of any auxiliary information that may be present on the screen. There will be the variable's name, the JavaBean used to edit variables of that type, along with a popup menu button to the left of the variable name. When the user moves the mouse over the variable name, a small Balloon Help style slip will popup and display the variable's one line description. Underneath the popup menu will be more specific commands that relate to the variable including the ability to open up the URL of the extended documentation in the browser, revert the variable to the default value for the study, get the value of a variable from a particular run, and get the value of the variable for a particular namelist. Other interface elements may appear on the screen for different layouts (see next section), but this succinct interface will be the one consistently available for the user.

Layout Engines

A *layout engine* is a piece of code that is responsible for taking a list of variables and their groups and creating a single screen or group of screens that lay them out in a programmatic fashion. They are similar to utilizing templates in a presentation program. The idea behind layout engines is that if a new layout format is desired, someone simply needs to code a new layout engine and it will appear everywhere in the program. There will be three default layout engines available in UniVista: verbose, sparse, and terse.

The *verbose* layout engine will display the most controls on the screen per variable. For each variable it will attempt to put on the screen the popup menu button, variable name, the entry area, the one-liner, a button to restore it to the study default, and a button linking to the external documentation. Each variable passed to the engine is laid out one underneath the other in a list fashion.

The *sparse* mode lays out variables one underneath each other including the variable name, popup menu button, entry area, and the one-liner. If there is enough room horizontally on the screen, variables will be laid out in two columns.

The *terse* mode lays out variables in one (or possibly two) columns using the standard layout of the popup menu button, the variable name, and entry area.

The layout engines are used in the study editor automatic screen creation wizards, and offer the potential to allow screens to be dynamically laid out during a user's run to allow them to use the style of interface they are most comfortable with.

Run Management

The run editor has the ability to create a session from one of the existing codes and studies and modify the values of the variables for a particular run of the simulation software. Runs can be created from a study, or by cloning an existing run. Runs can also be created by specifying a study and a namelist which is used to import values for the variables. Each run is stored in the database underlying UniVista in order to archive each simulation. Variable values and output files are stored. When each run is archived, the person who executed the run can determine whether all users should be able to see the run or whether it should be accessible to that user only.

UniVista Implementation

The first version of UniVista to arise out of this design is implemented on the Java 1.1.x platform using a database for storing all of UniVista specific information.

Storing all of the data in a database allows for easy creation of a secure, collaborative environment. Multiple users can connect to the same database and share codes, studies, and runs. This type of environment can become even more useful when codes are spread across laboratories, allowing easy sharing of information via the Internet. At the same time, the database provides security to UniVista. Only users with the proper database access privileges will be able to get at the data and modify it. This way, sensitive information can be kept private and modifications can be prevented to specific data. The database provides centralized data management and allows UniVista to easily be used in a multi-user environment.

Java was chosen as the implementation language of first version of UniVista for multiple reasons. One of the most important was platform independence. In the scientific community, different labs usually standardize around different brands of computers. In the PPPL alone there is a wide mixture of Macintosh, Wintel, and Solaris machines. Java provides the only way to run a program independently on each type of machine. The need for cross-platform compatibility also motivated the use of the older 1.1 version of the VM. Java 1.2, while providing many features that are very desirable from a programming perspective, is still not available on all platforms. If UniVista is to be immediately useful, it must therefore be written for the 1.1 VM.

Another benefit to Java is the advent of the Java Database Connectivity (JDBC) standard. This standard provides an interface to which the object based JDBC database drivers must conform. This allows the database code in UniVista to be independent of the driver. Because the driver requires only a string for identification, the user can install and use a new driver without needing to change the code of UniVista. Initially UniVista is using the JDBC-ODBC driver provided by Sun for getting ODBC connectivity on all platforms, but there is nothing preventing the use of a different driver for a specific database, or a more efficient bridge from a third party vendor.

It was chosen to use Swing 1.0.x to implement the user interface component of UniVista. Swing and its lightweight components provide truly platform independent user interfaces. In a heterogeneous environment, users would prefer to use a similar looking interface on all platforms. In addition, since UniVista is a user interface design tool, the user interfaces that are created should look the same on all platforms. One of the problems in the AWT is precisely that the UI is implemented using the primitives of the platform, resulting in the appearance (and sometimes size) of many UI elements being platform dependent. Swing draws its own components using the Java graphics architecture, and thus promises to be identical on all platforms which, barring certain font spacing problems in 1.0.x, it fulfills. In addition, the appearance of Swing can be customizable to match the platform at hand or be platform independent without needing to modify the layout. This provides a comfortable environment for the user, letting the user choose whether to use the cross platform appearance or the one tailored for their platform. The choice to use Swing 1.0.x instead of 1.1 is due to the need for compatibility with the 1.1 VM. The newer Swing, while completed (and unfortunately having its package renamed to javax), is not backwards compatible with other versions of Java.

Another benefit to Java is its reusable component model, JavaBeans. One of the biggest uses of JavaBeans has been to implement user interface components. This allows UI components to adhere to a somewhat standard interface as well as allowing code to examine these objects to determine their interface. This is quite beneficial for UniVista. By storing all of the UI elements as JavaBeans, their state can be easily saved and restored through object serialization. In addition, it provides the user with the flexibility to replace UI elements in the program by simply adding the JavaBean's jar file to their classpath and then instructing UniVista to load that class to edit a specific variable type. This potentially allows savvy users to provide new UI elements for complex types, such as a visual graphing JavaBean for arrays, without needing to rewrite UniVista. While this implementation detail is quite beneficial, it will not be implemented fully in the first version of UniVista. The inherent problem is that the different versions of Swing prior to 1.1 are not compatible via object serialization. This removes one of the largest benefits to JavaBeans. If time permits, a JavaBean wrapper for Swing JavaBeans may be developed and the full JavaBean model implemented.

The database design of UniVista is fairly simple. It consists of sets of indexed tables to store information about variable types, codes, screens, and runs. Throughout the tables the basic structure is to store the internal name as the key of a record, the short description, and then a serialized Java object that contains all of the information. While this means that it is difficult for users to change interfaces directly by modifying the database, it eases the implementation from the Java end as whole studies and interfaces can be stored with one call. The key name and description are intended to allow advanced users to muck with the databases if they desire, but this practice will not be encouraged. The table structure is to have one table containing all of the variable types; one global table contains all of the codes. Each code has its own table to contain all of the studies derived for that code, and each study has a table containing all of the runs performed with that study.

On a code level, UniVista is split up into three main packages. The UniVista.gui package contains all of the interface code for UniVista. The UniVista.database package contains the code used to map the database structures to the Java structures. The UniVista.support package contains the code used to provide support for storing and manipulating UniVista objects.

The initial implementation of this design is focusing on getting a functional product. The necessary elements of the design for a functional product are the ability to import and export namelists, design the codes and studies, and perform runs. To this end, only the basics of what would like to be supported for graphical design and variables are being implemented. This essentially includes everything except for the consistency checks and the complete storage of variables through serialized JavaBeans. Only the standard layout engine is being implemented as well, and only for the study editor and creation of initial studies from code variable lists. An object framework is being put in place for extending the layout engines to support dynamic creation of new layouts for users at the run level as well as to create new styles of layout engines. Other aspects that will remain unimplemented in the initial version of UniVista are drag and drop capabilities due to the lack of support for them in Java 1.1 and aliasing of variables.

Figure 1: Database Selection Dialog

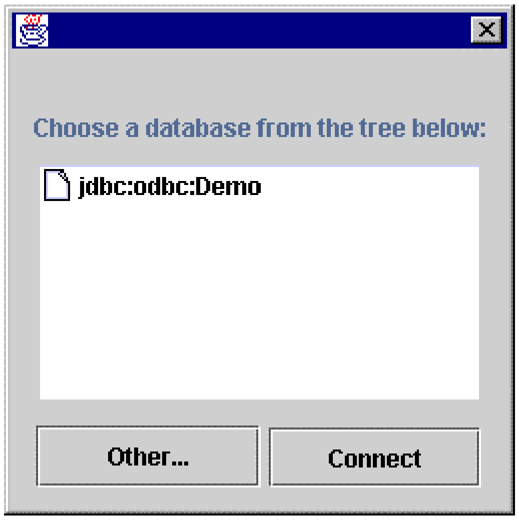


Figure 2: Database Login Dialog

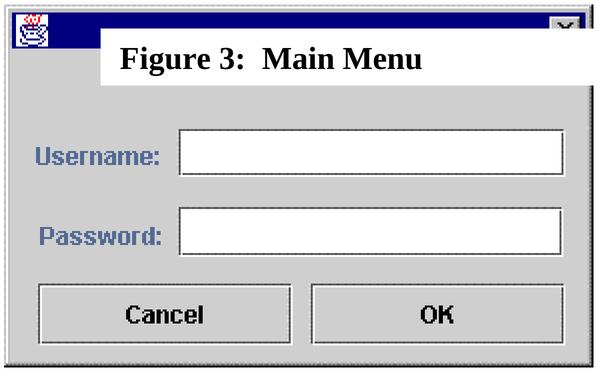


Figure 3: Main Menu

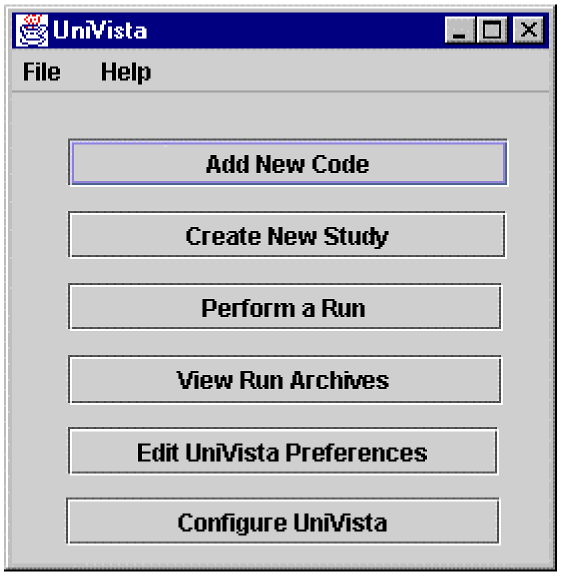


Figure 4: Code Editor

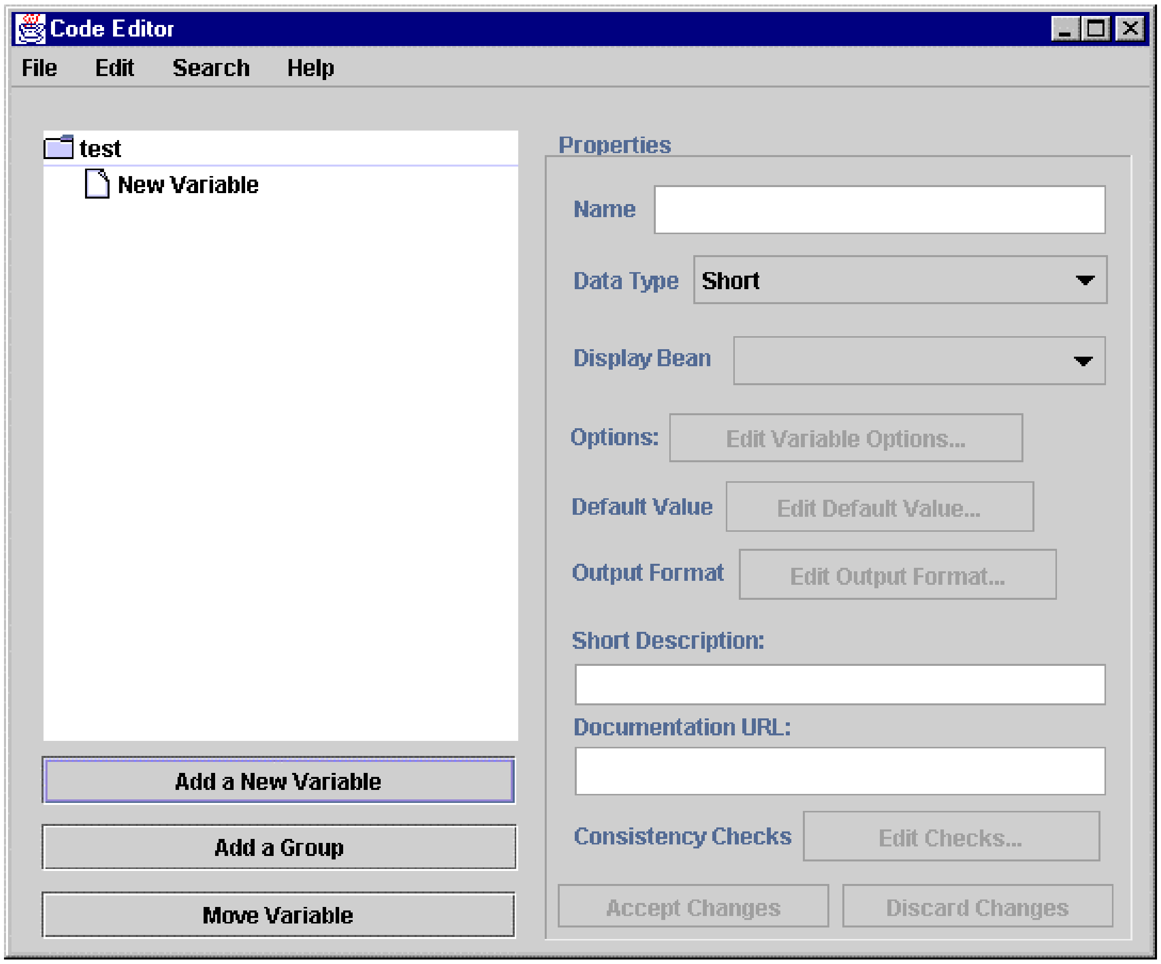
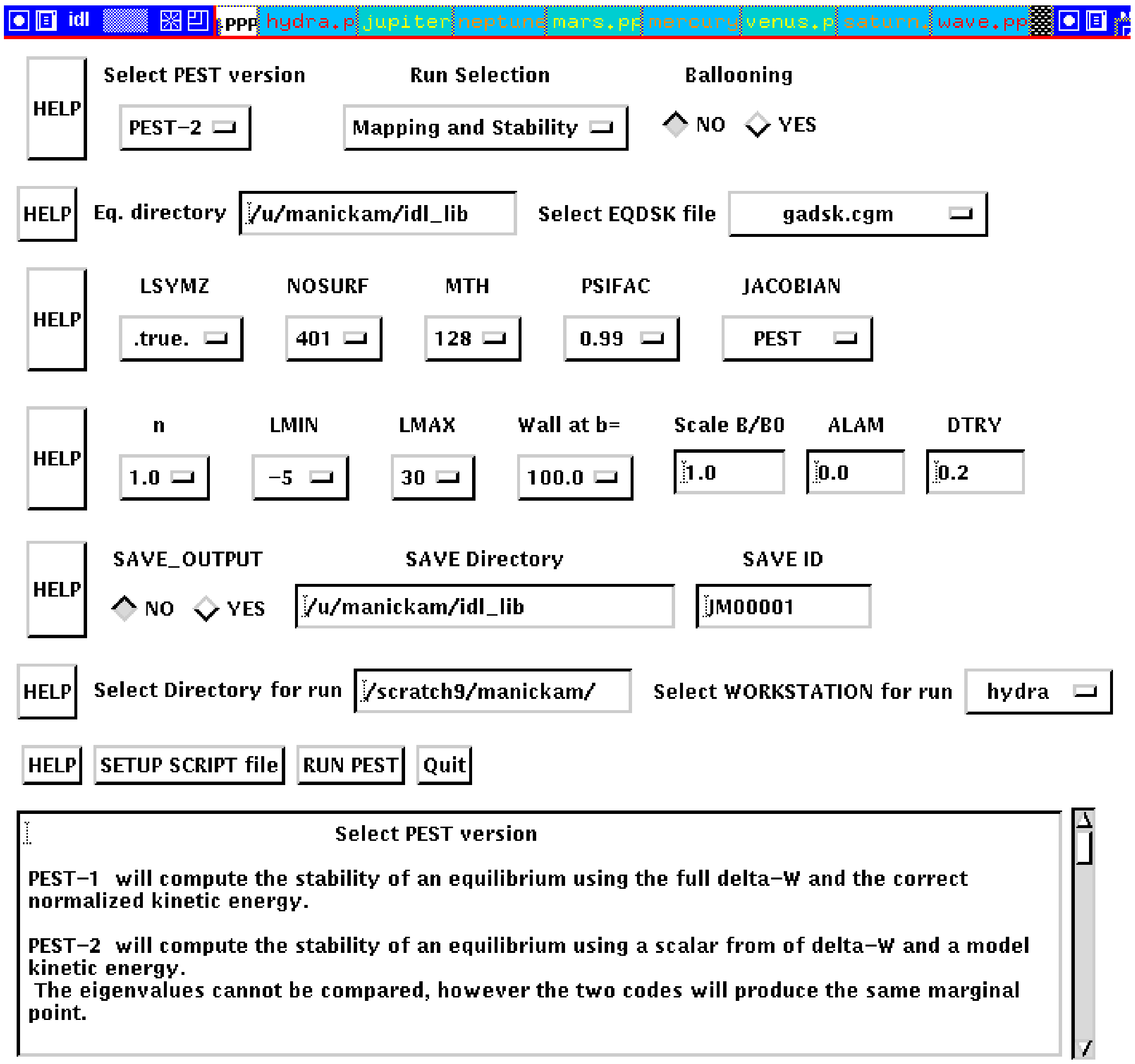


Figure 6: PEST GUI



Select PEST version: PEST-2

Run Selection: Mapping and Stability

Ballooning: [✓ NO] YES

Eq. directory: /u/manickam/idl_lib

Select EQDSK file: gadsk.cgm

LSYMZ	NOSURF	MTH	PSIFAC	JACOBIAN
.true.	401	128	0.99	PEST

n	LMIN	LMAX	Wall at b=	Scale B/B0	ALAM	DTRY
1.0	-5	30	100.0	1.0	0.0	0.2

SAVE_OUTPUT: [✓ NO] YES

SAVE Directory: /u/manickam/idl_lib

SAVE ID: JM00001

Select Directory for run: /scratch9/manickam/

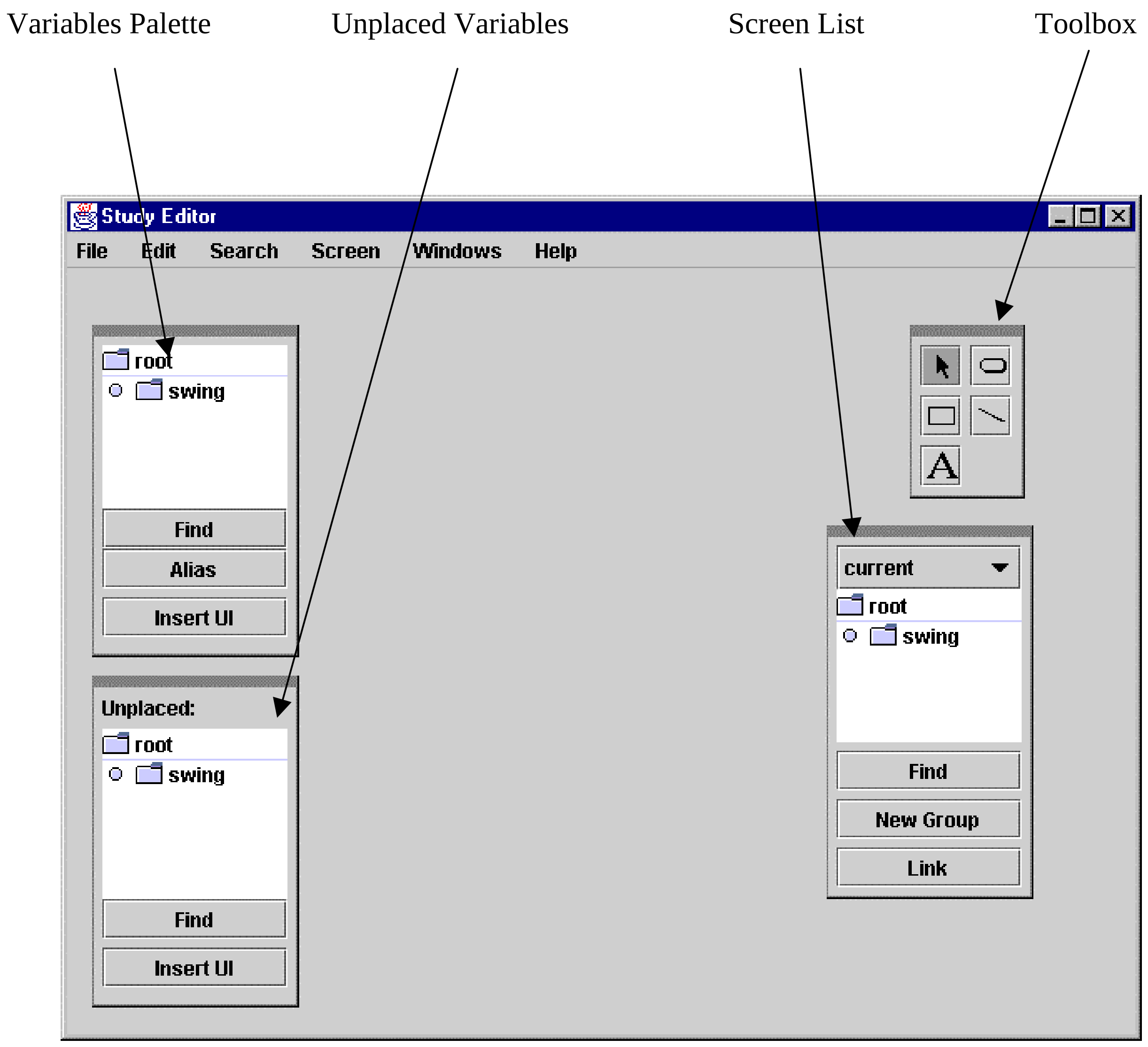
Select WORKSTATION for run: hydra

Select PEST version

PEST-1 will compute the stability of an equilibrium using the full delta-W and the correct normalized kinetic energy.

PEST-2 will compute the stability of an equilibrium using a scalar from of delta-W and a model kinetic energy.
The eigenvalues cannot be compared, however the two codes will produce the same marginal point.

Figure 5: Study Editor


Variables Palette
Unplaced Variables
Screen List
Toolbox
Study Editor
File
Edit
Search
Screen
Windows
Help
root
swing
Find
Alias
Insert UI
Unplaced:
root
swing
Find
Insert UI
current
root
swing
Find
New Group
Link